

# Compilers

*Dr. Sherin ElGokhy*  
*Lecture#3*

# Implementation of Lexical Analysis

# Outline

- Specifying lexical structure using regular expressions
- Finite automata
  - Deterministic Finite Automata (DFAs)
  - Non-deterministic Finite Automata (NFAs)
- Implementation of regular expressions  
RegExp  $\Rightarrow$  NFA  $\Rightarrow$  DFA  $\Rightarrow$  Tables

# Notation

- How regular expressions are used to construct a full lexical specification on a programming language?
- At least one:  $A^+$   $\equiv AA^*$
- Union:  $A \mid B$   $\equiv A + B$
- Option:  $A + \varepsilon$   $\equiv A?$
- Range:  $'a' + 'b' + \dots + 'z'$   $\equiv [a-z]$
- Excluded range:  
complement of  $[a-z]$   $\equiv [^a-z]$

# Regular Expressions in Lexical Specification

- A specification for the predicate

$$s \in L(R) \quad \text{?????}$$

- What we want to do?
- We want to know whether a given string is in this language or not.
- But a yes/no answer is not enough!
- Instead: partition the input string into tokens. Each one of these tokens is in  $L(R)$ .
- We adapt regular expressions to this goal

# Regular Expressions => Lexical Spec. (1)

## Steps to construct a full lexical specification from regular expressions

1. Write a **rexp** for the lexemes of each token
  - Number = **digit** +
  - Keyword = **'if' + 'else' + ...**
  - Identifier = **letter (letter + digit)\***
  - OpenPar = **'('**
  - ...

# Regular Expressions => Lexical Spec. (2)

2. Construct  $R$ , matching all lexemes for all tokens

$$R = \text{Keyword} + \text{Identifier} + \text{Number} + \dots$$
$$= R_1 + R_2 + \dots$$

- The union of all the regular expressions forms the lexical specification of a language?

# Regular Expressions => Lexical Spec. (3)

3. Let input be  $x_1 \dots x_n$   
For  $1 \leq i \leq n$  check  
$$x_1 \dots x_i \in L(R)$$
4. If success, then we know that  
$$x_1 \dots x_i \in L(R_j) \text{ for some } j$$
5. Remove  $x_1 \dots x_i$  from input and go to (3)

Repeat until the input string is empty (the entire input is analyzed)



# Ambiguities (1)

- There are ambiguities in the algorithm
- How much input is used? What if
  - $x_1 \dots x_i \in L(R)$  and also
  - $x_1 \dots x_K \in L(R)$
- Rule: Pick longest possible string in  $L(R)$ 
  - The “maximal munch”

# Ambiguities (2)

- Which token is used?

What if

- $x_1 \dots x_i \in L(R_j)$

and also

- $x_1 \dots x_i \in L(R_k)$

- Rule: use rule listed first (j if  $j < k$ )
  - Treats “if” as a keyword, not an identifier

# Error Handling

- What if  
No rule matches a prefix of input ?
- Problem: Can't just get stuck ...Compilers should do good error handling.
- Solution:
  - Write a rule matching all “bad” strings  
**ERROR=[all strings not in the lexical specification]**
  - Put it last (lowest priority)

# Summary

- Regular expressions provide a concise notation for string patterns
- Use in lexical analysis requires small extensions
  - To resolve ambiguities
  - To handle errors
- Good algorithms known
  - Require only single pass over the input
  - Few operations per character (table lookup)

# Finite Automata

- Regular expressions = specification
- Finite automata = implementation mechanism for regular expressions
- A finite automaton consists of
  - An input alphabet  $\Sigma$
  - A finite set of states  $S$
  - A start state  $n$
  - A set of accepting states  $F \subseteq S$
  - A set of transitions  $\text{state} \xrightarrow{\text{input}} \text{state}$

# Finite Automata

- Transition

$$s_1 \xrightarrow{a} s_2$$

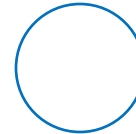
- Is read

In state  $s_1$  on input "a" go to state  $s_2$

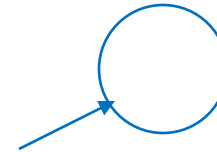
- If end of input and in accepting state => accept
- Otherwise => reject

# Finite Automata State Graphs

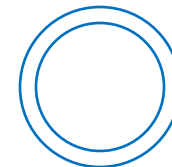
A state



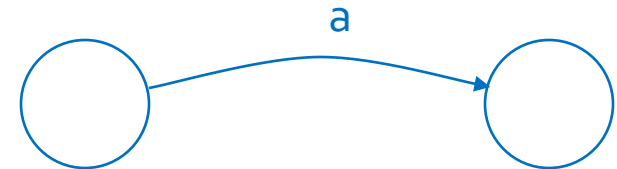
The start state



An accepting state

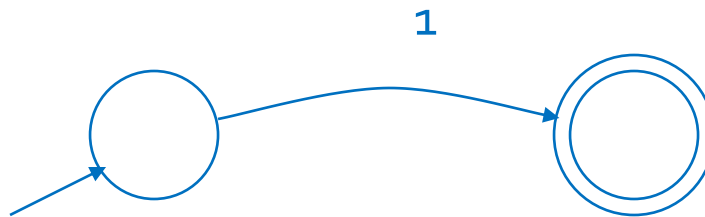


A transition



# A Simple Example

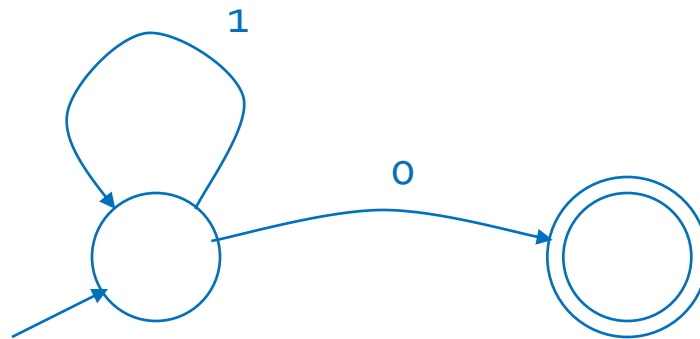
- A finite automata that accepts only "1"





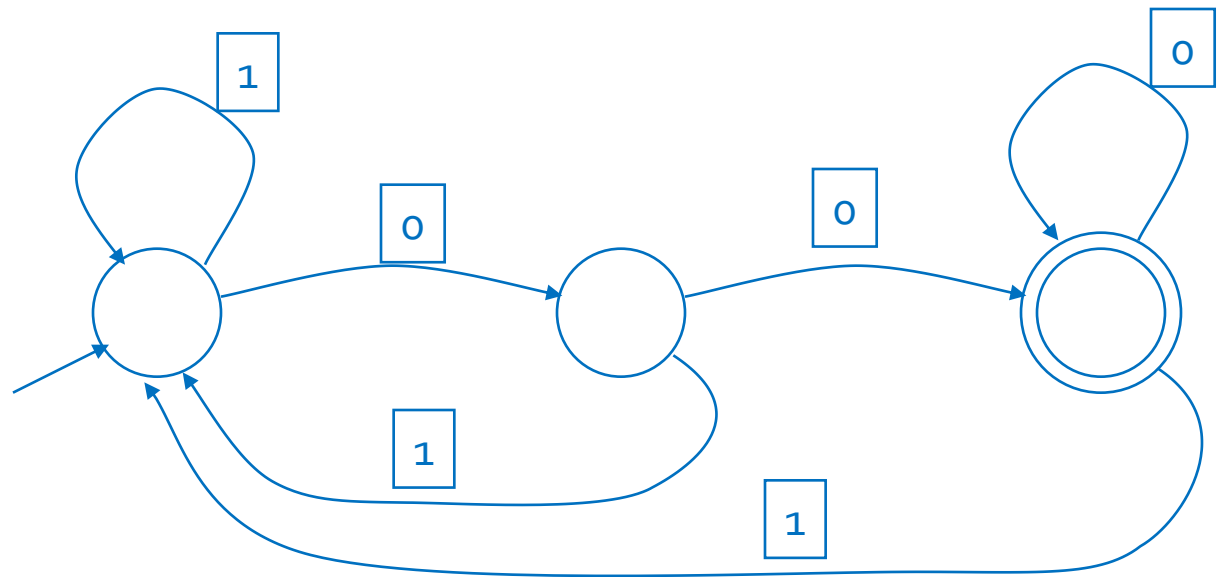
## Another Simple Example

- A finite automaton accepting any number of 1's followed by a single 0
- Alphabet: {0,1}



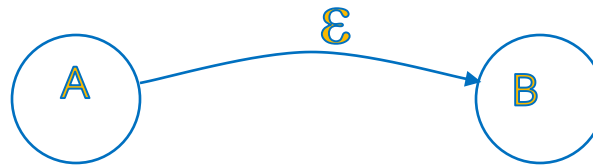
## Another Example

- Alphabet  $\{0,1\}$
- What language does this recognize?



# Epsilon Moves

- Another kind of transition:  $\epsilon$ -moves



- Machine can move from state **A** to state **B** without reading input
- Think of  $\epsilon$ -moves as a kind of free move.

# Deterministic and Non- deterministic Automata

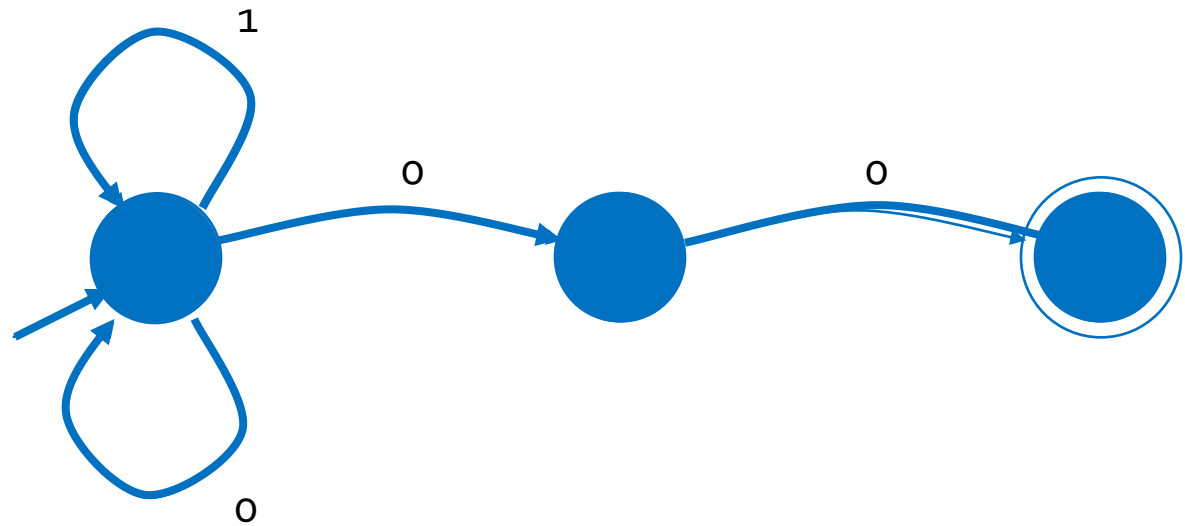
- Deterministic Finite Automata (DFA)
  - One transition per input per state
  - No  $\epsilon$ -moves
- Nondeterministic Finite Automata (NFA)
  - Can have multiple transitions for one input in a given state
  - Can have  $\epsilon$ -moves

# Execution of Finite Automata

- A DFA can take only one path through the state graph
  - The path is completely determined by the input (The machine has no choice)
- NFAs can choose
  - Whether to make  $\epsilon$ -moves
  - Which of multiple transitions for a single input to take

# Acceptance of NFAs

- An NFA can get into multiple states



- Input: 1 0 0

Rule: NFA accepts if it can get to a final state

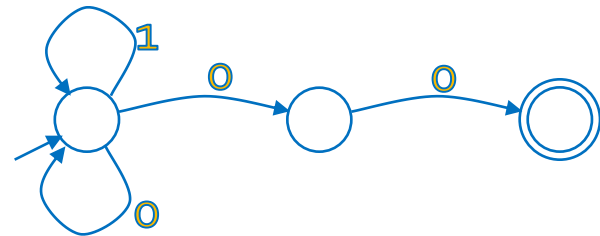
## NFA vs. DFA (1)

- NFAs and DFAs recognize the same set of languages (regular languages)
- DFAs are faster to execute
  - There are no choices to consider

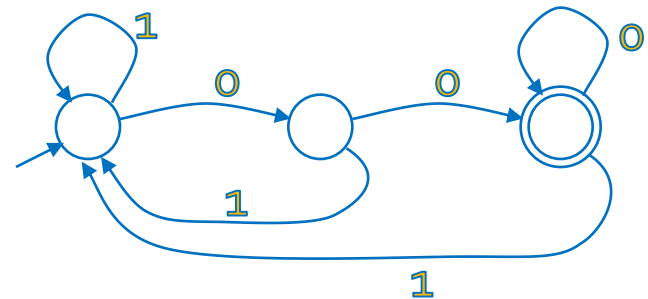
## NFA vs. DFA (2)

- For a given language NFA can be simpler than DFA

NFA



DFA

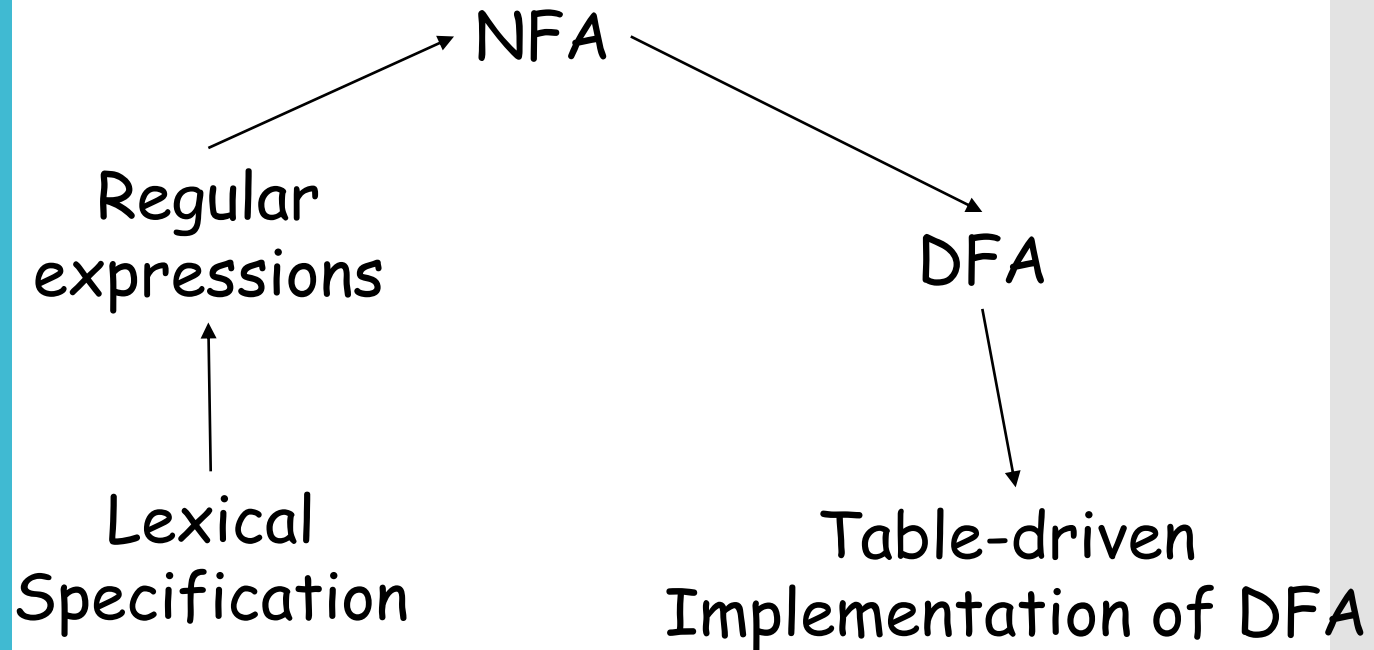


DFA can be exponentially larger than NFA



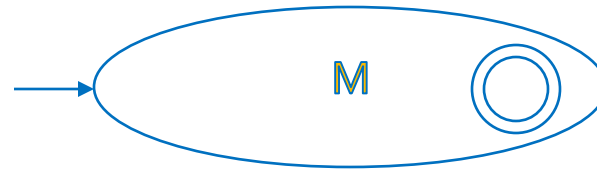
# Regular Expressions to Finite Automata

- High-level sketch



# Regular Expressions to NFA (1)

- For each kind of rexp, define an NFA
  - Notation: NFA for rexp **M**



- For  $\epsilon$

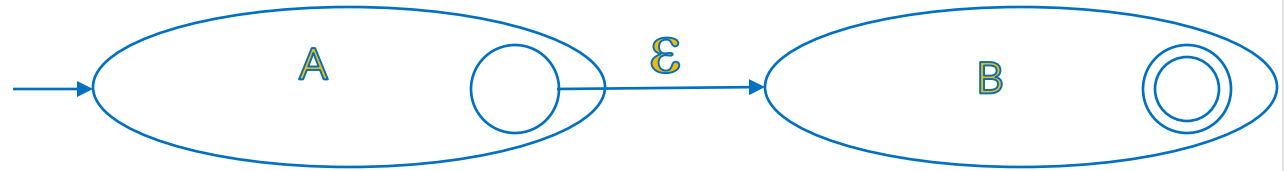


- For input **a**

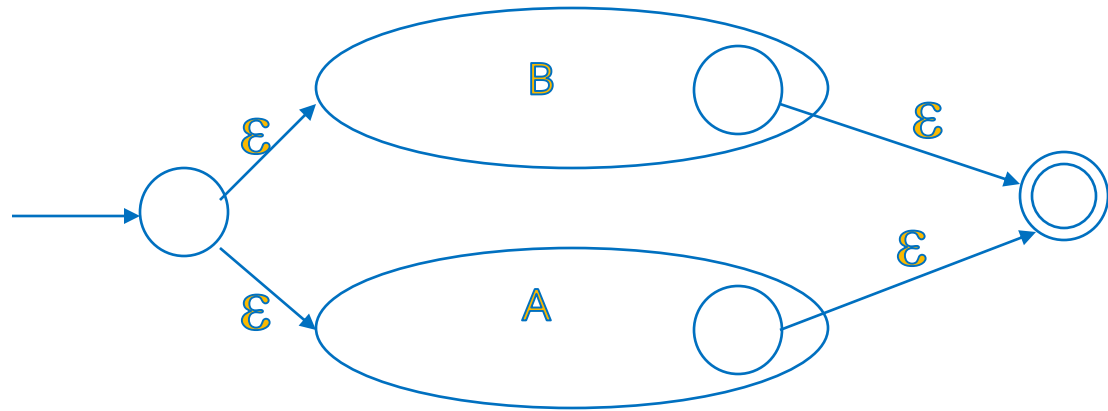


# Regular Expressions to NFA (2)

- For  $AB$

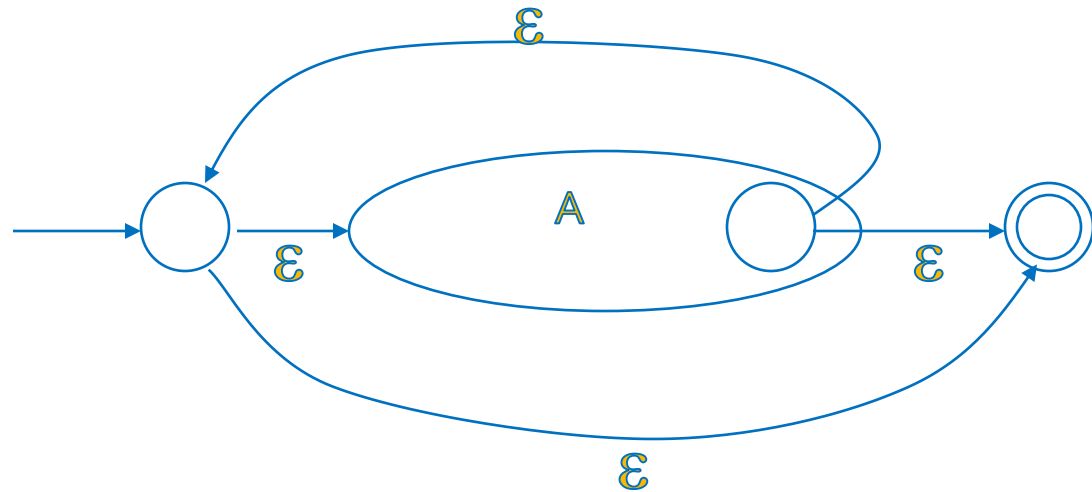


- For  $A + B$



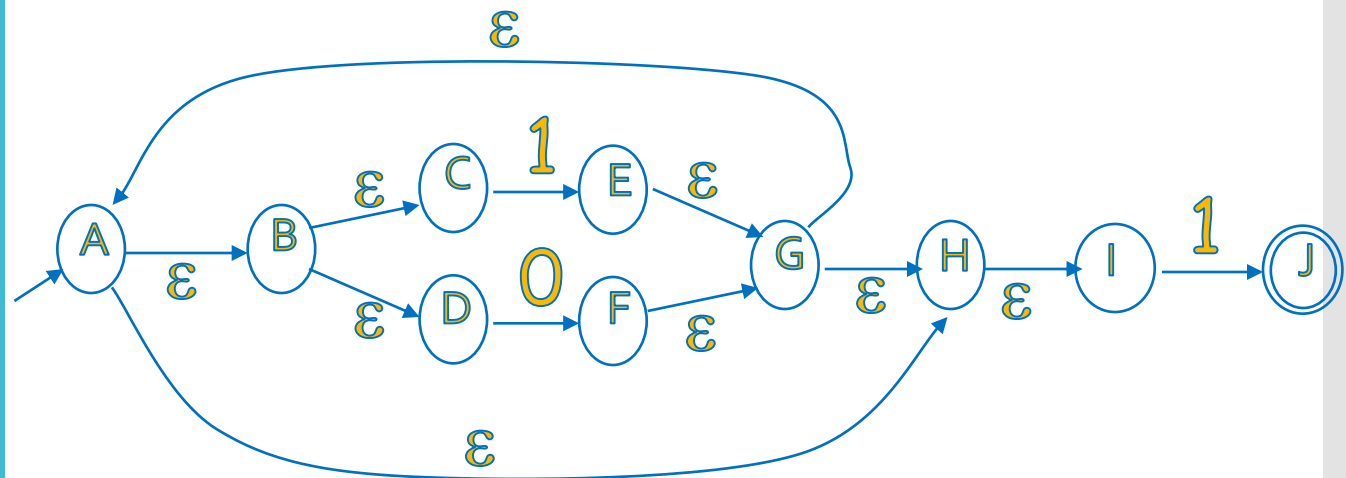
# Regular Expressions to NFA (3)

- For  $A^*$



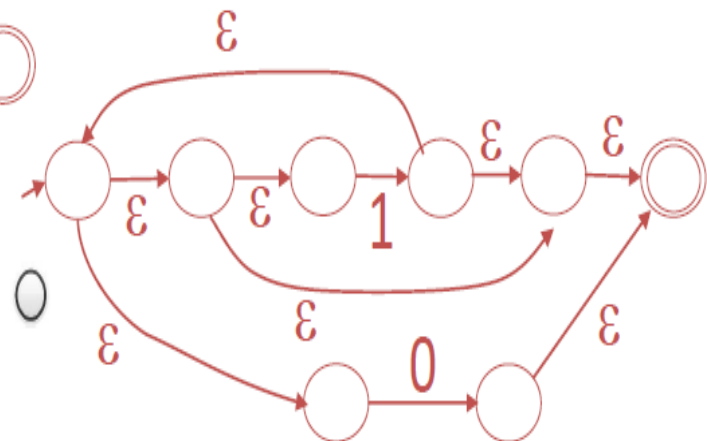
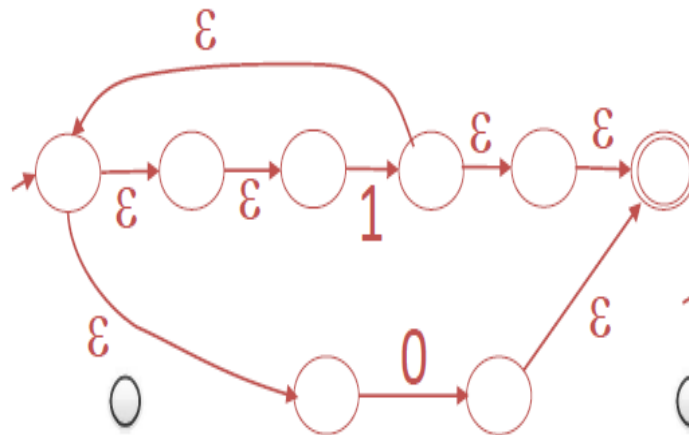
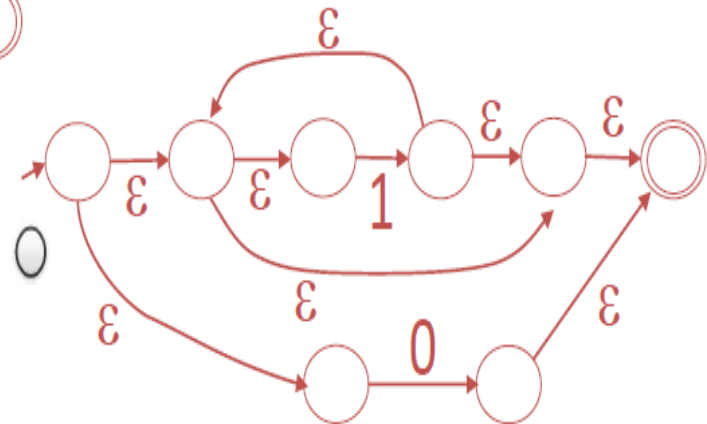
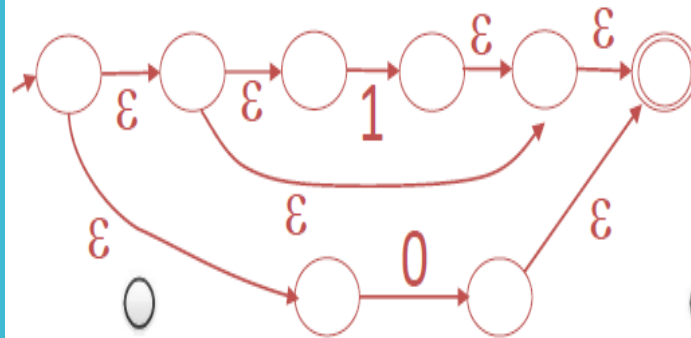
# Example of RegExp -> NFA conversion

- Consider the regular expression  
 $(1+0)^*1$
- The NFA is

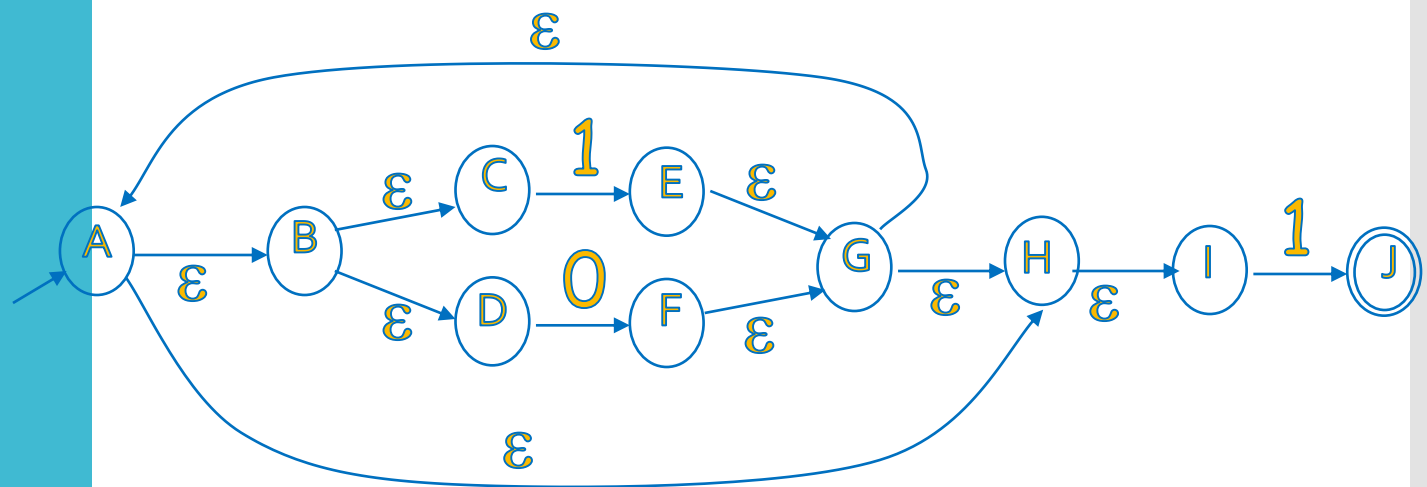


# Quiz

Choose the NFA that accepts the following regular expression:  $1^* + 0$



NFA to  
DFA.



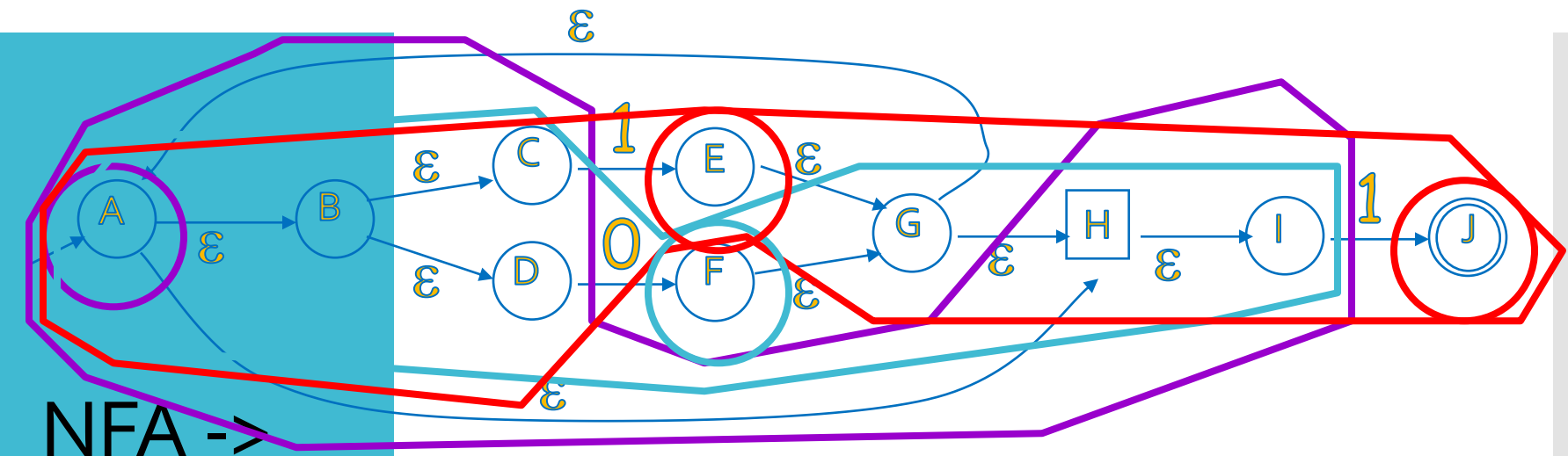
## NFA to DFA: *The Trick*

- Simulate the NFA
- Each state of DFA  
= a non-empty subset of states of the NFA
- Start state  
= the set of NFA states reachable through  $\epsilon$ -moves from NFA start state
- Add a transition  $S \xrightarrow{a} S'$  to DFA iff
  - $S'$  is the set of NFA states reachable from any state in  $S$  after seeing the input  $a$ , considering  $\epsilon$ -moves as well

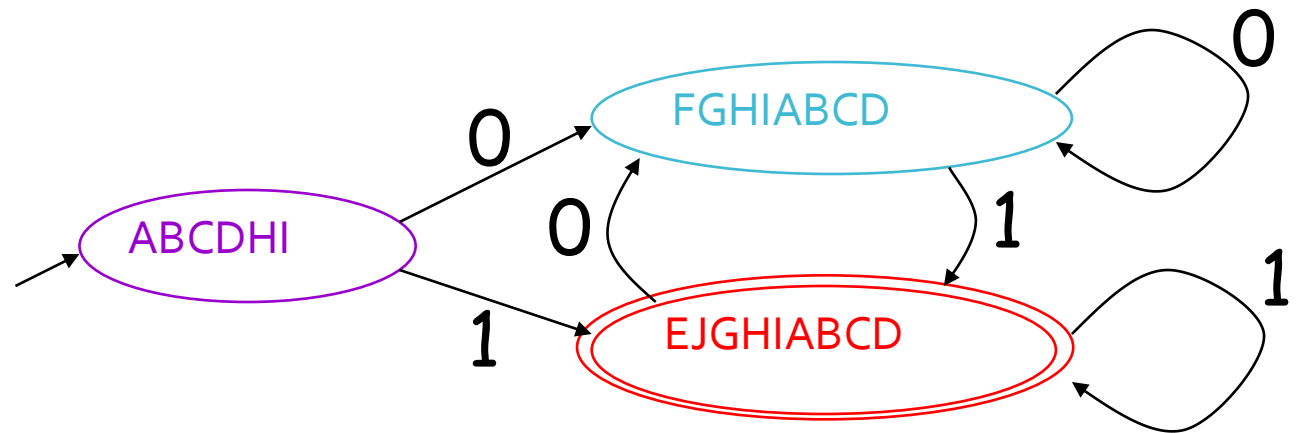


## NFA to DFA. Remark

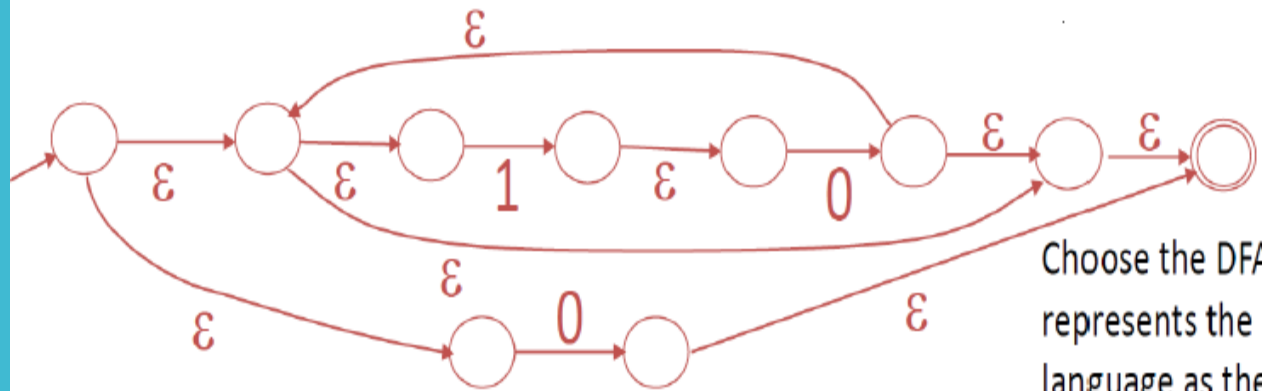
- An NFA may be in many states at any time
- How many different states ?
- If there are  $N$  states, the NFA must be in some subset of those  $N$  states
- How many subsets are there?
  - $2^N - 1 =$  finitely many



NFA  $\rightarrow$   
DFA  
Example

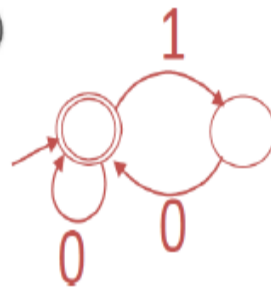


# QUIZ

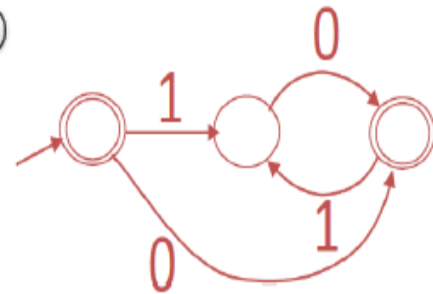


Choose the DFA that represents the same language as the given NFA

☐



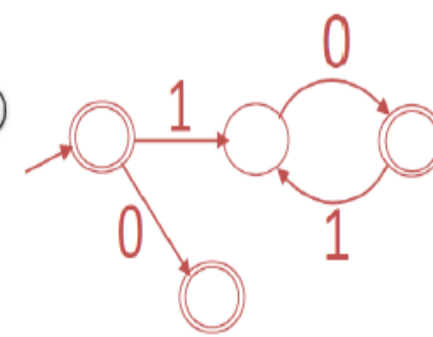
☐



☐



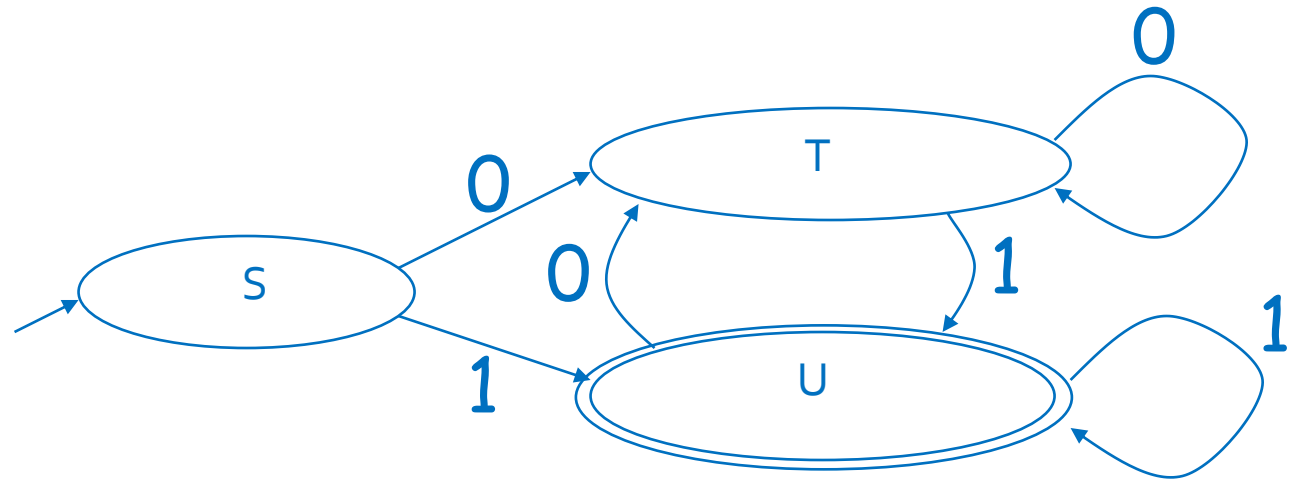
☐



# Implementation

- A DFA can be implemented by a 2D table  $T$ 
  - One dimension is “states”
  - Other dimension is “input symbol”
  - For every transition  $S_i \xrightarrow{a} S_k$  define  $T[i,a] = k$
- DFA “execution”
  - If in state  $S_i$  and input  $a$ , read  $T[i,a] = k$  and skip to state  $S_k$
  - Very efficient

# Table Implementation of a DFA



	0	1
S	T	U
T	T	U
U	T	U

## Implementation (Cont.)

- NFA -> DFA conversion is at the heart of tools such as flex
- But, DFAs can be huge
- In practice, flex-like tools trade off speed for space in the choice of NFA and DFA representations
- DFA : faster, less compact
- NFA: slower, consumes less memory

*Thanks*